Warm up exertcise

3 Review: Memory Errors

- Where are the memory errors/leaks in this code?

```
int *func(int x) {
  int *y = new int(x);
  y = new int[x];
  return y;
}

int main() {
  int *a = func(5);
  int *b = a;
  delete b;
  cout << a[2] << endl;
}
```

Memory orphaned here.
Watch out! These look a lot alike!
Wrong delete for array.
Using deleted memory.

video 1: RAII: a strategy for managing Dynamic resources

5 Example: Memory Leak

- This function allocates two arrays.
  - However, it does not use the `delete` operator.
  - Although the pointers `arr` and `arr2` are cleaned up when they go out of scope, the arrays they point to are not.

```
void doSomeArrayThing(int size) {
  int *arr = new int[size];
  arr[0] = 5;

  int *arr2 = new int[2 * size];
  arr2[0] = 5;
}
```

- But wait, why can't the compiler add this rule?

"When a pointer goes out of scope…
…just delete what it points to."

①指向 dynamic memory 的 pointer die了，并不代表着 dynamic memory 中存储的 objects 也被 delete 了.
②思考为什么 pointer 不能在 die 时自动 delete 所指 object.

6 Pointers and the Compiler

- Why can't the compiler clean up the array for you?
- **Proposal**: *Whenever a pointer object goes out of scope, the compiler deletes the object it points to.*

```
void doSomeArrayThing(int size) {
A  int *arr = new int[size];
B  arr[0] = 5;

C  int *arr2 = new int[2 * size];
D  arr2[0] = 5;
}
```

- **Exercise**: Find a way to break this scheme by adding or modifying one line of code above.

Note: The STL provides "smart pointers", allow you to opt-in to something like this for certain pointers. They're pretty cool, but beyond the scope of the course (for now).

3/5/2021

①两种情境，推翻之前的设想.

7 Example: Managing Dynamic Arrays

- General principle: *Prefer to package new/delete inside an ADT rather than use them manually.*

```
void doSomeArrayThing(int size) {
  // Allocate memory
  int *arr = new int[size];
  int *arr2 = new int[2 * size];

  // Use it
  arr[0] = 5;
  arr2[0] = 5;

  // Free memory
  delete[] arr;
  delete[] arr2;
}
```

```
void doSomeArrayThing(int size) {
  // Constructor - Create ADTs
  // Memory allocated automatically
  DynamicIntArray arr(size);
  DynamicIntArray arr2(2 * size);

  // Use it
  arr[0] = 5;
  arr2[0] = 5;

  // Free memory?
```

①将"new"和"delete" package 进 class 的 ctor 和 dtor 中.

8 Example: Managing Dynamic Arrays

```
class DynamicIntArray {
private:
  int *arr;
  int arr_size;

public:
  DynamicIntArray(int size)
    : arr(new int[size]), arr_size(size) { }

  ~DynamicIntArray() {
    delete[] arr;
  }

  int size() const {
    return arr_size;
  }

  int & operator[](int i) { return arr[i]; }
  const int & operator[](int i) const { return arr[i]; }
};
```

Constructor allocates a dynamic array.
Destructor cleans up the dynamic array.
Member functions and operators make up the interface.
The const version can be used for DynamicIntArrays declared as const.

3/5/2021

如何初始化 array 在 heap 中.
实用 pointer 来接收 来自 heap 的地址
通过 "int[size]" 告诉 compiler 是个 array & array 的数据类型 & Capacity.

10 RAII: Resource Acquisition Is Initialization

- Problem:
  The compiler is bad at managing resources.
  - If we use `new` to create dynamic memory, the compiler can't clean it up for us.
- Observation:
  The compiler can manage the lifetime of local objects.
  - Automatic storage duration.
  - Cleaned up when they go out of scope.
- Idea:
  Wrap up resource management in a class and use local instances of the class to access the resources.
  - **Ctor:** The resource is acquired when the object is born.
  - **Dtor:** The resource is released when the object dies.

①利用 compiler 对于 local variable "storage duration" 的控制，实现对 resources 的 management.

13 Using a Dynamic Array Instead

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;

  int capacity;

  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow();
};
```

Instead of directly having the array as a member, we use a pointer to a dynamically allocated array.
The lifetime of the array is now independent from the UnsortedSet.
Current capacity of the dynamic array. This no longer has to be the same for all UnsortedSets.
Number of elements in the set. (Number of valid cells in the array.)

①每个 set 的 capacity 可不相同
②如何实现不断 grown 的 set.

14 UnsortedSet Destructor

```
template <typename T>
class UnsortedSet {
public:
  UnsortedSet()
    : elts(new T[DEFAULT_CAPACITY]),
      capacity(DEFAULT_CAPACITY),
      elts_size(0) {}

  ~UnsortedSet() {
    delete[] elts;
  }
  ...

private:
  T *elts;
  int capacity;
  int elts_size;
};
```

Allocate dynamic array in the constructor. elts points to it.
Clean up memory for dynamic array in the destructor.

①private member 须要的 3 个 variable
②ctor 须要的初始化
③dtor 须要的删除
④如何 new/delete array in dynamic memory.

15 Destructors

- We used a destructor to take care of this in `DynamicIntArray`.
- What is the role of the destructor?
  - Common misconception:
    The destructor destroys the object…No!
  - It gives the object a chance to put its affairs in order before dying.
  - In the two examples today, this means cleaning up the dynamic array so it isn't leaked.

①Destructor 的定义
②Destructor 和 dynamic memory 之间的联系.

16 Calling grow() as Needed

```
template <typename T>
class UnsortedSet {
public:
  void insert(T v) {
    if (contains(v)) { return; }
    if (elts_size == capacity)) { grow(); }

    elts[elts_size] = v;
    ++elts_size;
  }

private:
  T *elts;
  int capacity;
  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow();
};
```

If we are attempting to add an element, but there is no more room, call grow() to allocate a larger dynamic array.

①grow() function 的使用情境.
②grow() 函数的安放位置 (private)

17 Exercise: grow()

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
  int capacity;
  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow() {
    // TODO: WRITE YOUR CODE HERE
  }
};
```

In order to grow…
1. Make a new array with twice as much capacity
2. Copy elements over
3. Update capacity
4. Destroy old array
5. Point elts to the new array

①grow() 的思路
②grow() 的 implementation.

21 Solution: grow()

```
template <typename T>
class UnsortedSet {
...
private:
  T *elts;
  int capacity;
  int elts_size;

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow() {
    T *newArr = new T[2 * capacity];
    for (int i = 0; i < elts_size; ++i) {
      newArr[i] = elts[i];
    }
    capacity *= 2;
    delete[] elts;
    elts = newArr;
  }
};
```

In order to grow…
1. Make a new array with twice as much capacity
2. Copy elements over
3. Update capacity
4. Destroy old array
5. Point elts to the new array

Video 2: growable containers

11 Upgrading UnsortedSet

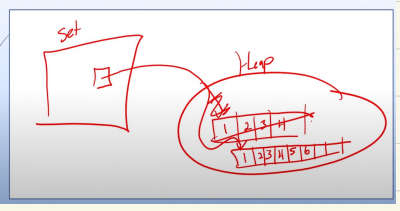
```
template <typename T>
class UnsortedSet {
public:
  // Maximum capacity of a set.
  static const int ELTS_CAPACITY = 10;

  ...

  // REQUIRES: size < ELTS_CAPACITY
  // EFFECTS:  adds v to the set
  void insert(T v);

  ...
};
```

Let's remove the fixed capacity restriction. We'll use dynamic memory in the implementation of UnsortedSet to make it work.

①去除 fixed capacity 的限制
②用 dynamic memory 来实现 unsortedSet.

12 Idea

- Dynamically allocate the array…
- Need more space? Make a new, larger array, copy over the elements, and throw away the old one!



①grow function 思路.

Video 3: dynamic resource invariants

22 The Power of Indirection

- In `UnsortedSet`, we've decoupled the lifetime of the object (i.e. the set itself) from the dynamic array it uses.
- This is possible because the array is not **directly** a member of the class.
- Instead, we use a pointer to work with the array **indirectly**.
  - We can just make a new one if we need it to be bigger (and free up the old one)

Note: The lifetime of the member `elts` itself is tied to the whole object, but `elts` is just the pointer, not the array!

3/5/2021

①使用指向 dynamic memory 的 pointer 来 indirectly 控制 array.
②与在 stake 中的 array 相比，在 heap 中的 array 不是 set class 的 member, pointer 才是.

23 new and delete

- Generally, it's a good strategy to think of matching up `new` and `delete` in your code.
  - Allocate with `new`, clean up with `delete`.
  - This doesn't mean you literally have the same number of `new` and `delete`.
  - A single `new`/`delete` can be used many times if it's in a loop, a function called several times, etc.
- All objects that come from a `new` must eventually meet a `delete`!
- Use this pattern for constructors and destructors.
  - If you use `new` in the constructor, almost certainly you need to use `delete` in the destructor.

①new 和 delete 数量不用相同.
②ctor 有 new, dtor 有 delete,

24 Dynamic Resource Invariant

```
template <typename T>
class UnsortedSet {
...
private:

  // INVARIANT: An UnsortedSet manages a dynamically
  //            allocated array. During its lifetime
  //            there is exactly one such array,
  //            pointed to by elts.
  T *elts;
  int capacity;
  int elts_size;

  int indexOf(int v) const {
    for(int i = 0; i < elts_size; ++i){
      if(elts[i] == v){ return i; }
    }
    return -1;
  }
};
```

Add an invariant to guarantee elts is always safe to use.
we need to make sure that the pointer points to an array which should not be null or uninitialized
Because elts is now just a pointer, do we have to worry it might be uninitialized or null?

ensure the constructor sets up this invariant and all of our function maintain that invariant so we can rely on it
①保证 pointer 指向的 heap 中的 array 是有效化过的，且不空的
②保证只有一个在 heap 中的 array.

25 Dynamic Resource Invariant

```
template <typename T>
class UnsortedSet {
...
private:
  // INVARIANT: An UnsortedSet manages a dynamically allocated
  // array. During its lifetime there is exactly one such array,
  // pointed to by elts.
  T *elts;

  UnsortedSet() : elts(new T[DEFAULT_CAPACITY]),
                  capacity(DEFAULT_CAPACITY), elts_size(0) {}

  ~UnsortedSet() { delete[] elts; }

  // Changes underlying representation to use a
  // dynamic array of 2 * capacity elements
  void grow() {
    T *newArr = new T[2 * capacity];
    for (int i = 0; i < elts_size; ++i) { newArr[i] = elts[i]; }
    capacity *= 2;
    delete[] elts;
    elts = newArr;
  }
};
```

Estabiiish invariant.
Clean up.
Restore invariant.
Break invariant (temporarily).
invariant

Exercise:

Exercise

Question
How many of these code snippets leak memory?
A) 0 B) 1 C) 2 D) 3 E) 4

```
template <typename T>
class UnsortedSet {
private:
  T *elts;

  UnsortedSet() : elts(new T[DEFAULT_CAPACITY]),
                  capacity(DEFAULT_CAPACITY), elts_size(0) {}

  ~UnsortedSet() { delete[] elts; }
};
```

```
void func() {
  UnsortedSet<int> s1;
  s1.insert(2);
  s1.insert(3);
}
```

```
void func() {
  UnsortedSet<int*> s2;
  s2.insert(new int(2));
  s2.insert(new int(3));
}
```

```
void func() {
  UnsortedSet<int> *s3
    = new UnsortedSet<int>;
  s3->insert(2);
  s3->insert(3);
}
```

```
void func() {
  UnsortedSet<int> *s4
    = new UnsortedSet<int>;
  s4->insert(2);
  s4->insert(3);
  delete s4;
}
```